Ficheros

Sistemas Operativos

Enrique Soriano

GSYC

23 de febrero de 2022

Ficheros

- Tradicionalmente: datos que persisten tras la ejecución del proceso y/o del sistema (excepción: ramfs), y que son compartidos por distintos procesos y/o sistemas.
- Requisitos:
  - Gran tamaño
  - Durabilidad
  - Acceso concurrente
  - Protección
- Más general: es una **interfaz**. P. ej. puede representar un dispositivo.

Ficheros

- Estructura: fichero simple (Unix), Resource Forks (Mac)...
- Nombre: ¿extensión?
- ¿Tipos?
- Ya hemos usado ficheros desde nuestros programas, p. ej. `printf()`.
- Operaciones básicas: open, read, write, close.

Discos: estructura interna

- Bloque físico $\rightarrow$ determinado por el HW (sector). Tradicionalmente de 512 bytes, algunos fabricantes están pasando a 4 Kb.
- Bloque lógico $\rightarrow$ determinado por el sistema de ficheros.
- Si no coinciden, empaquetamos.
- Siempre hay fragmentación interna en los bloques.
- ¿Tamaño? compromiso entre:

$$\uparrow \text{tamaño de bloque} \Rightarrow \uparrow \text{fragmentación}$$
$$\uparrow \text{tamaño de bloque} \Rightarrow \uparrow \text{tasa de transferencia}$$

Discos

Acceso al disco:

- Secuencial: se accede registro a registro (contiguos).
- Aleatorio: se accede a cualquier registro inmediatamente.

¿Y los discos duros actuales? Caches flash, discos SSD, híbridos, etc.

¿Hay alguno secuencial hoy en día?
`https://en.wikipedia.org/wiki/IBM_3592`

Discos

- Direccionamiento de bloques:
  - CHS: Cylinder, Head, Sector. Esquema viejo.
  - LBA: Linear.
- Así *eran*:

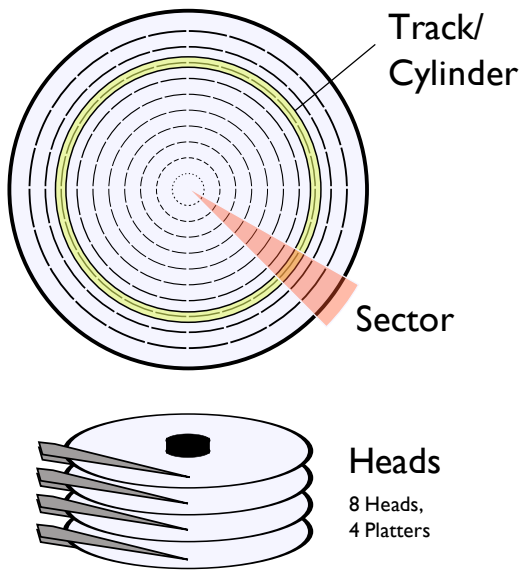


Imagen: Public Domain, Wikipedia

Recordemos: particiones

- Un disco se puede dividir en partes: particiones.
- Esquema BIOS o UEFI.
- Podemos tener volúmenes lógicos (p. ej. LVM en Linux).
- Imágenes de discos: iso, bin, dmg, etc...
- En linux, los comandos `fdisk` y `parted` (o su variante gráfica, `gparted`) permite manipular las particiones de un disco.

Discos

- Algoritmo de planificación de disco: ¿cómo organiza el SO las operaciones?
- El algoritmo depende totalmente del tipo de disco que usemos.
- ¿Cómo se relaciona el direccionamiento LBA con la geometría real del disco? Se sigue suponiendo que sí. ¿Se puede generalizar? No. ¿Hay caches internas en el disco? ¿Es un disco de verdad o es un disco virtual?
- Los clásicos:
  - FIFO: es justo, no altera el orden de las operaciones. Lento en discos mecánicos.
  - Shortest Seek First: no es justo, puede provocar hambruna, desordena operaciones.
  - Ascensor: compromiso justicia/eficiencia, no provoca hambruna, desordena operaciones.
  - ... (hay muchos otros)

Sistemas de Ficheros

- El sistema de ficheros nos ofrece la abstracción de *fichero*: no pensamos en los bloques, para nosotros un fichero es una secuencia de bytes.
- ¿Cómo asigna los bloques a un fichero?
- Se puede implementar de distintas formas: asignación continua, enlazada, enlazada con tabla, indexada

Asignación de espacio: contigua

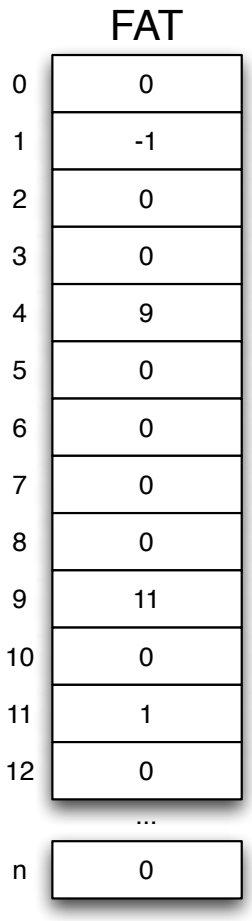
- Un fichero ocupa una serie de bloques contiguos en disco.
- Acceso rápido en discos de acceso secuencial.
- Rendimiento alto: un acceso para conseguir un dato cualquiera $\rightarrow$ la entrada de directorio contiene la dirección de inicio y la longitud del fichero.
- Problema: si hay asignación dinámica $\rightarrow$ fragmentación externa.
- Problema: hay que saber el tamaño máximo del fichero cuando se crea.
- Tamaños limitados por los huecos $\rightarrow$ compactación.
- ¿Se usa? Sí.

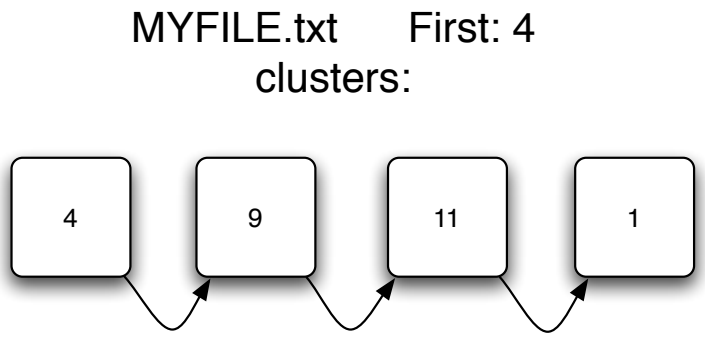
Asignación de espacio: enlazada con tabla

FAT es un caso de asignación de espacio usando una **lista enlazada** implementada con una tabla:

- Usa bloques lógicos grandes: **clusters**.
- La tabla FAT tiene una entrada por cluster.
- El índice de la tabla es el número de cluster.
- La entrada de directorio tiene la referencia a la entrada primer cluster.
- La entrada del cluster actual referencia la entrada del siguiente cluster.
- Mejora del acceso aleatorio respecto a una lista sin tabla: no hay que leer los clusters del fichero para conseguir la dirección del cluster N.

FAT


FAT
0
0
1
-1
2
0
3
0
4
9
5
0
6
0
7
0
8
0
9
11
10
0
11
1
12
0
...
n
0


MYFILE.txt First: 4
clusters:
4
9
11
1

FAT

- Si queremos la tabla FAT en memoria principal $\rightarrow$ no puede ser muy grande $\rightarrow$ clusters grandes $\rightarrow$ fragmentación interna.
- El acceso aleatorio es más rápido que en lista enlazada normal.
- ¿Y si se estropea la tabla FAT?

Ejemplo: FAT32

- El sector 0 de la partición es el *boot sector* (también está duplicado en el sector 6). Contiene código del cargador secundario e información sobre el sistema de ficheros:
  - Nº de sectores, 4 bytes (Max. tam. de disco: 2 Tb)
  - Etiqueta del volumen.
  - Nº de copias de la tabla FAT.
  - Primer cluster del raíz.
  - ...

Ejemplo: FAT32

- Entrada de directorio (32 bytes):
  - Nombre del archivo, 8 Bytes.
  - Extensión del archivo, 3 Bytes.
  - Atributos del archivo, 1 Byte.
  - Reservado, 10 Bytes.
  - Hora de la última modificación, 2 bytes.
  - Fecha de la última modificación, 2 bytes.
  - Primer cluster del archivo, 4 bytes.
  - Tamaño del archivo, 4 bytes.
- Clusters de 32 Kb, 64 sectores.

Asignación de espacio: indexada

- Idea: se indexan los bloques de datos del fichero en un *bloque de indirección*.
- Bloque de indirección grande $\rightarrow$ desperdicio.
- Bloque de indirección pequeño $\rightarrow$ no soporta ficheros grandes.
- Para un acceso a datos, siempre son dos accesos a disco.

Asignación de espacio: indexada multinivel

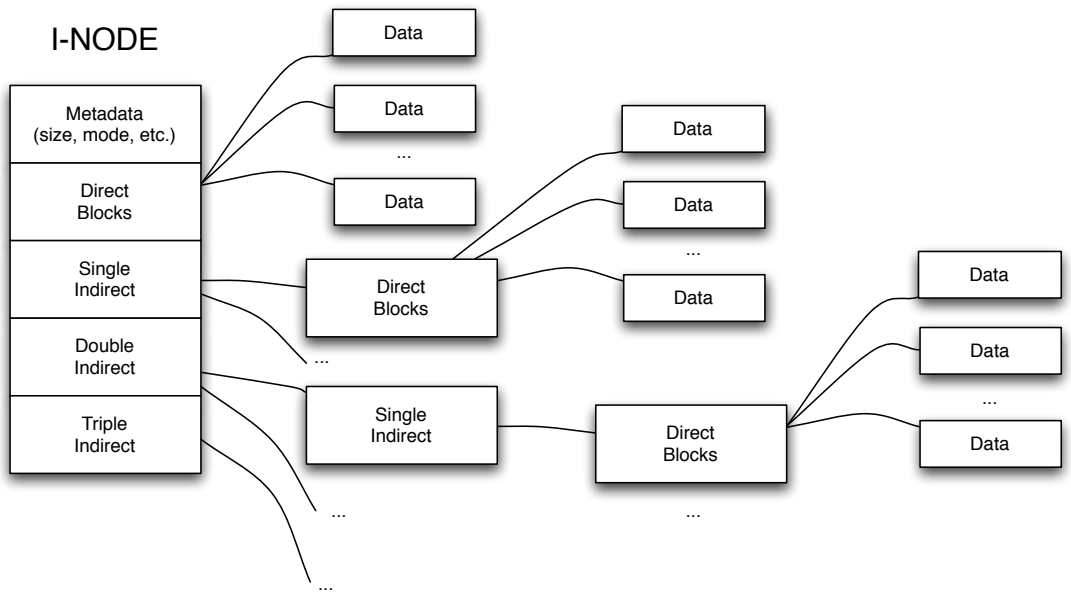
- Por niveles. P. ej. indirección doble:
  - Los bloques de índice de 1er nivel apuntan a bloques de índice de 2º nivel.
  - Los bloques de índice de 2º nivel apuntan a bloques de datos.
- Ejemplo: bloques de 4Kb con punteros de 4 bytes.
- Problema: Para los ficheros pequeños, desperdiciamos.
- Problema: Para un acceso a datos, n accesos a disco.

Asignación de espacio: indexada con esquema combinado

- Algunos bloques directos de datos.
- Algunos bloques de indirección simple.
- Algunos bloques de indirección doble.
- ...
- Nos quedamos con lo mejor de cada modelo anterior.

Ficheros en Unix: i-nodos

Es un caso de asignación indexada con esquema combinado:



Unix: Partición

- Bloque de arranque (boot block): código de arranque del sistema.
- Superbloque: tamaño del volumen, número de bloques libres, lista de bloques libres, tamaño del vector de i-nodos, siguiente i-nodo libre, cierres...
- Vector de i-nodos: representación de los ficheros.
- Bloques de datos.

Unix: i-nodos

- Cada fichero/directorio tiene un i-nodo asociado, definido por un número de i-nodo. El directorio raíz siempre tiene el número de i-nodo 2.
- La estructura se localiza en el vector indexando por el número de i-nodo.
- El OS mantiene una cache de i-nodos en memoria.

Unix: i-nodos

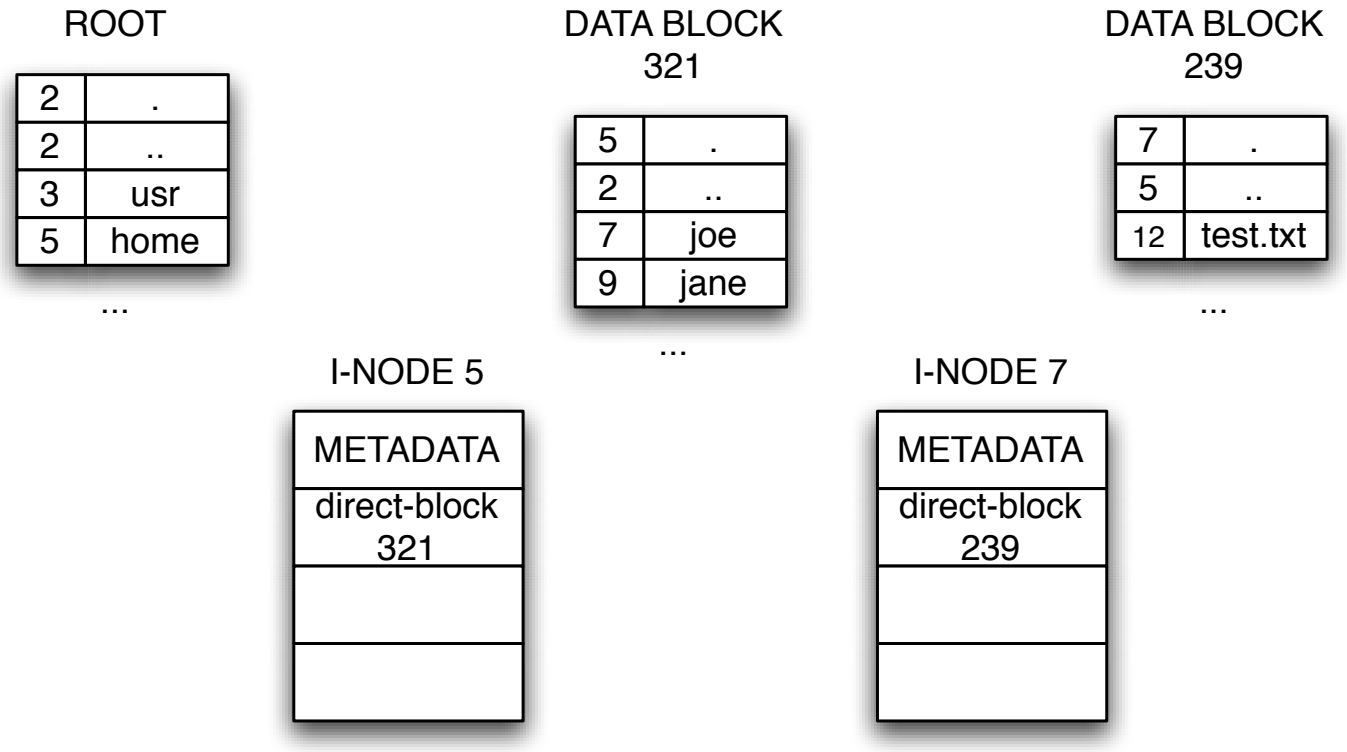
- Recuerda lo que tiene un i-nodo:
  - permisos
  - tiempos
    - acceso a datos (`atime`)
    - modificación de los datos (`mtime`)
    - modificación del i-nodo (`ctime`)
  - tamaño
  - dueño
  - tipo
  - número de bloques
  - contador de referencias (links)
- **No** contiene el nombre de fichero.

Unix: entradas de directorio

- Un directorio relaciona un nombre con un i-nodo: *entrada de directorio* (dentry).
- Un directorio tiene:
  - Su propio i-nodo
  - Bloques de datos con la lista de entradas de directorio.
- Entre las entradas, tiene la entrada de . (su i-nodo) y .. (i-nodo del padre).
- El kernel mantiene una cache de entradas de directorio en memoria.

Unix: ejemplo

/home/joe/test.txt



Unix: enlaces

Recuerda:

- Enlaces duros (hard links): es otro nombre para el fichero, la entrada de directorio apunta al mismo i-nodo que el antiguo nombre. En general, no se permite crear enlaces duros para directorios: rompen la jerarquía, crean bucles y crea ambigüedad en dot-dot.
- Enlaces simbólicos (symbolic links): es un fichero cuyos datos contienen la ruta al fichero enlazado $\rightarrow$ pueden *romperse*.

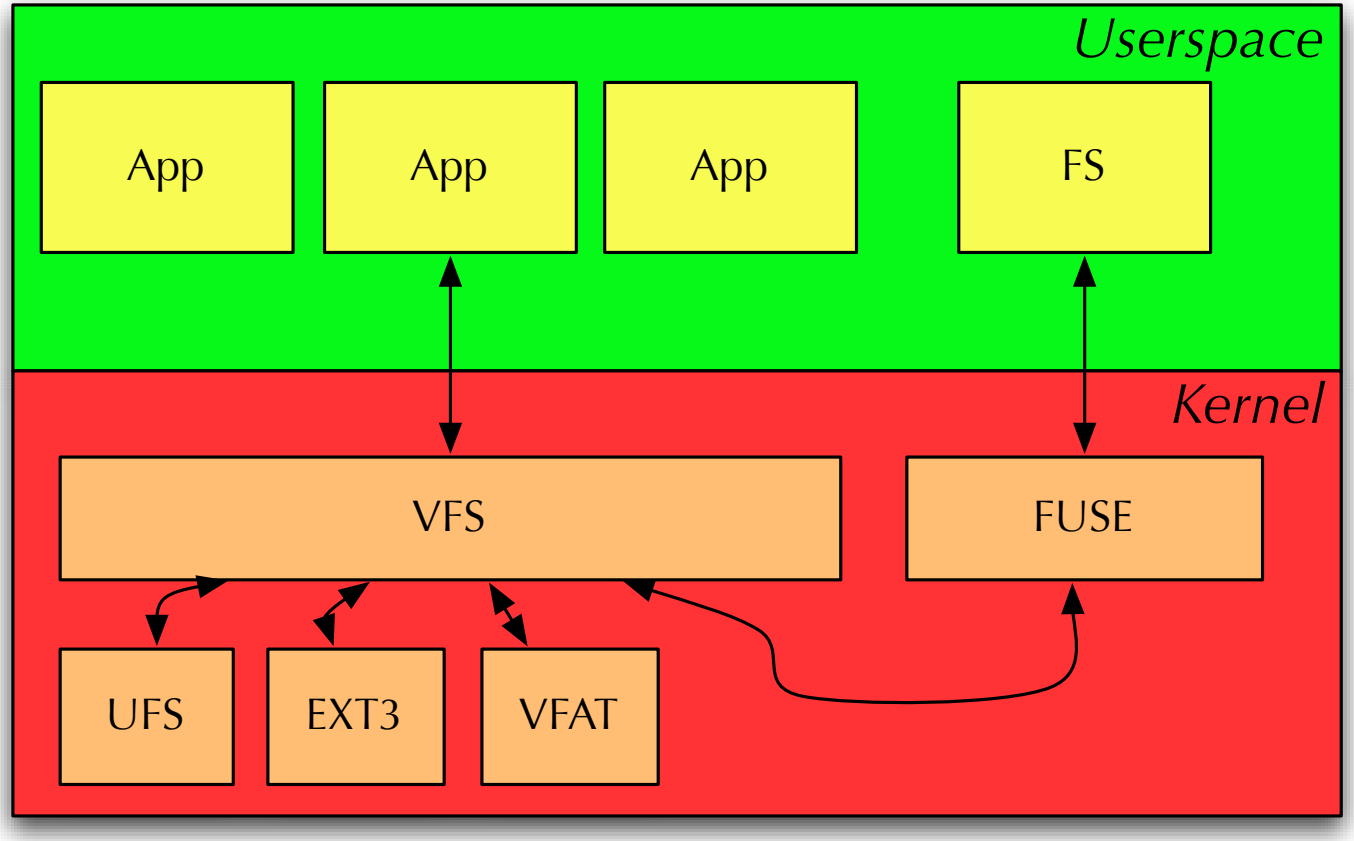
Gestión de espacio libre

- Bitmap: rápido. Inconveniente: puede ser demasiado grande como para tenerlo entero en memoria .P. ej. para 2TB de bloques $\rightarrow$ mapa de 512 MB.
- Lista enlazada de bloques libres: ineficiente si buscamos varios.
- Agrupaciones: el primer nodo libre en la lista no está vacío; tiene dentro enlaces a n bloques libres, los n-1 primeros vacíos, el último con n más. Ventaja: se pueden encontrar rápidamente las direcciones de n bloques libres.
- Cuenta: guarda en una lista el bloque libre y el número de bloques contiguos libres que le siguen.

Sistemas de Ficheros: implementación

- VFS (Virtual File System): interfaz común con el kernel para todos los sistemas de ficheros.
- Lo normal es que el sistema de ficheros esté implementado como un módulo del kernel.
- Pero también puede estar implementado como un programa de espacio de usuario.
- FUSE: módulo del kernel para implementar sistemas de ficheros en espacio de usuario que se puedan integrar con VFS.

Sistemas de Ficheros: implementación



Cache

- Ya lo veremos en el tema de memoria: el kernel mantiene una cache en RAM para los ficheros.
- Es muy eficiente. P. ej. La *buffer cache* de 4.4BSD evita el 85 % de los accesos a disco/red para conseguir bloques.
- En ocasiones es necesario saltársela (e.g. modo `O_DIRECT` de `open` en Linux).
- Cuando el sistema se está quedando sin marcos de página, se suele sacrificar espacio de la cache.
- Esto complica el problema de los fallos: ¿qué pasa si se el sistema falla y los datos están en la cache?
- El comando y llamada al sistema `sync` de UNIX sincroniza la cache con el disco y actualiza el superbloque. Se ejecuta periódicamente.

Fallos

- Una pérdida de coherencia en los metadatos (estructuras del sistema de ficheros) **es peor** que perder los datos.
- El comando `fsck` sirve para reparar el sistema de ficheros después de un fallo.
- Hay que recorrer los bloques de los ficheros para detectar errores, p. ej.:
  - Un bloque está asignado a un fichero y en la lista de bloques libres a la vez.
  - Un bloque en dos ficheros a la vez.
- Esto es caro.
- Los sistemas de ficheros modernos usan técnicas para no acabar con metadatos incoherentes: **journaling**.

Journaling

Ideas básicas:

- Las operaciones se escriben en un *journal* (en disco) de forma **atómica y ordenada** usando el soporte hardware del disco.
- Una vez escritas en el *journal* las operaciones están comprometidas.
- En algún momento las operaciones se aplicarán en el FS.
- Si el sistema falla, al rearrancar se aplican todas las operaciones pendientes en el *journal* (si las hay) para recuperar la coherencia.
- Desventaja: hay que escribir los datos dos veces en el disco (pero en el *journal* es secuencial).
- Se puede hacer *journaling* sólo de los metadatos, o de los metadatos y los datos.

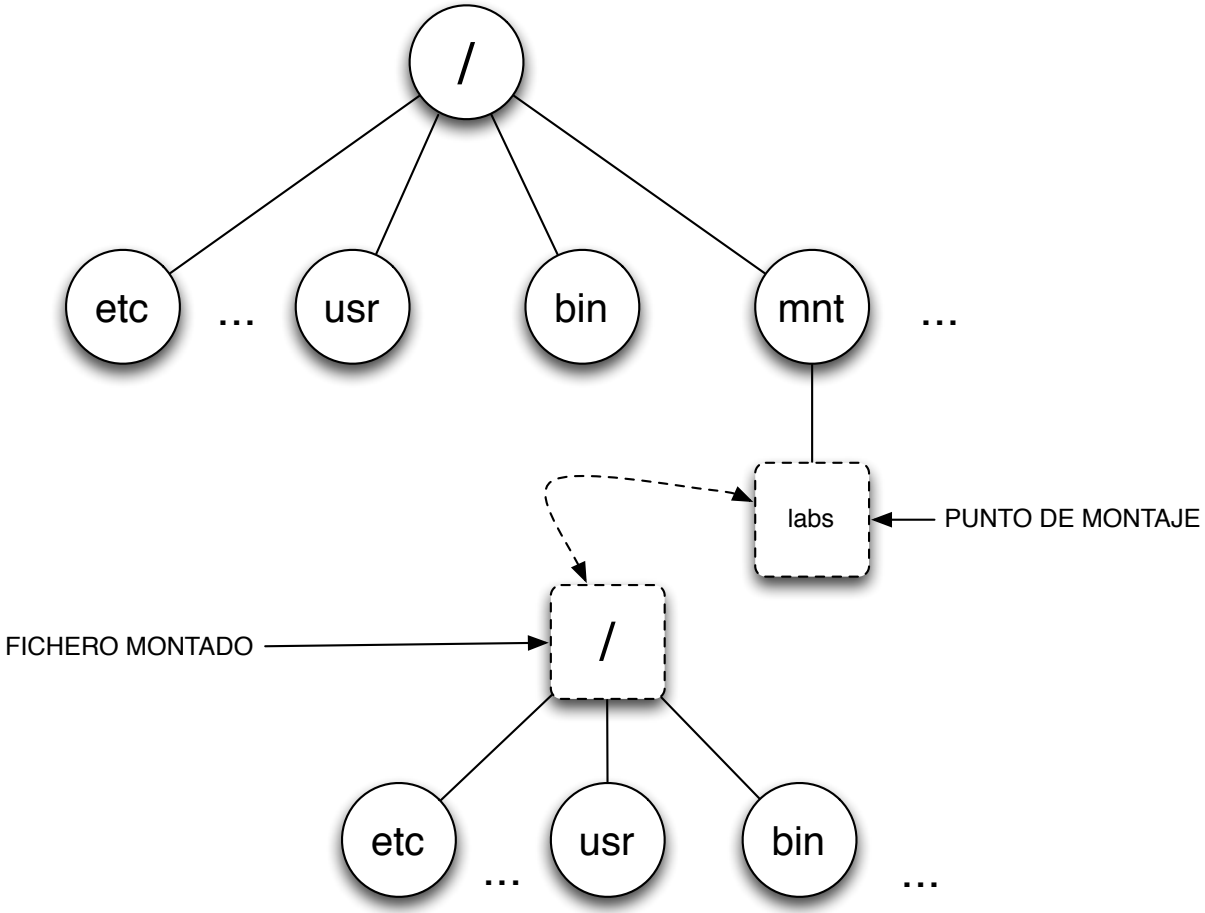
Espacio de nombres

- El árbol de ficheros del sistema.
- Se “camina” (walk) por la ruta, resolviendo cada parte.
- Se pueden *montar* nuevos árboles al espacio de nombres.
- El comando `mount` nos permite ver y modificar el espacio de nombres.
- Unix: un espacio de nombres común para todos los procesos del el sistema.
- Linux y otros sistemas modernos dejan tener diferentes espacios de nombres a grupos de procesos.

Espacio de nombres

Ejemplo:

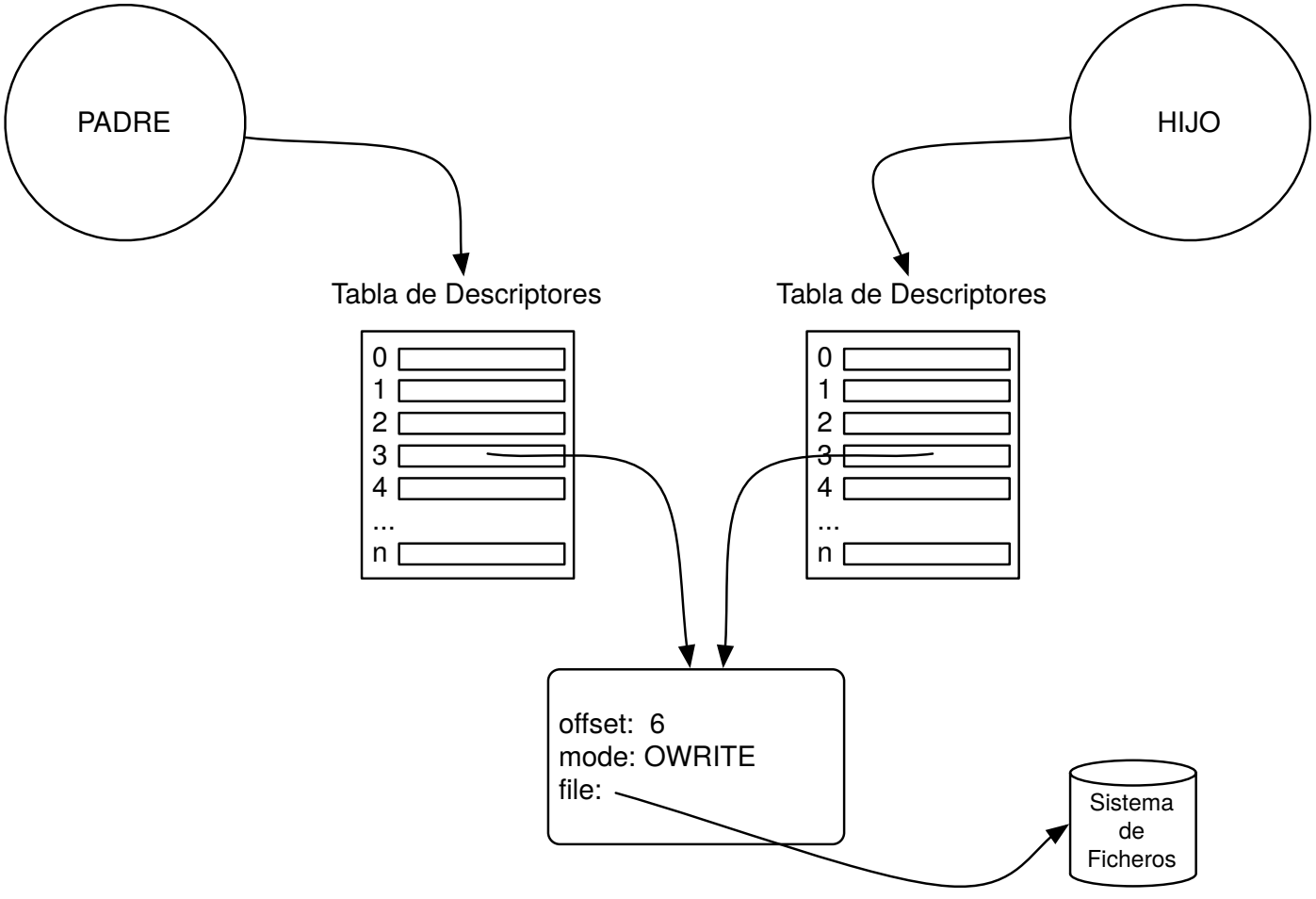
```
sshfs pepe@alpha.aulas.gsyc.urjc.es:/ /mnt/labs
```



Procesos y ficheros

- Por cada proceso, el kernel mantiene una tabla con los ficheros que tiene abiertos: **tabla de descriptores de fichero**.
- Un fichero abierto tiene un **descriptor de fichero (file descriptor)** que lo identifica.
- El número de descriptor de fichero es el índice en la tabla.
- Un fichero abierto tiene offset, modo de apertura, ...
- Los hijos heredan una copia de la tabla del padre.
- Hay tres posiciones especiales en la tabla: (0) entrada estándar, (1) salida estándar, y (2) salida de errores.

Descriptores de fichero



Terminal

- Los procesos que creamos en el shell normalmente tienen como entrada, salida y salida de errores al **terminal**.
- No siempre es así, y se puede cambiar.
- El terminal es un fichero como cualquier otro.
- Cuando se lee, se lee del teclado.
- Cuando se escribe, se escribe en la pantalla.
- `/dev/tty*` son los terminales, `/dev/pts/*` son pseudotermianles (terminales emulados).
- Para un proceso, su terminal siempre se llama:

  `/dev/tty`

Read

- `read`: lee como mucho el número de bytes indicado del fichero abierto en ese descriptor. Puede leer menos bytes sin ser un error: **lectura corta**. Si devuelve 0 bytes, se ha llegado al final del fichero.

```
ssize_t read(int fd, void *buf, size_t count);
```

Write

- `write`: escribe el número de bytes del buffer indicado en el fichero abierto en ese descriptor. Si devuelve un número distinto al número solicitado, se debe considerar un error.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Pread/Pwrite

- Hay versiones de read y write a las que se le da el offset:

```
ssize_t pread(int d, void *buf,
              size_t nbyte, off_t offset);

ssize_t pwrite(int fildes, const void *buf,
              size_t nbyte, off_t offset);
```

Lseek

- `lseek`: cambia el offset del descriptor de fichero. Retorna el offset resultante, -1 en error. Tiene tres modos de operación (tercer parámetro):
  - SEEK_SET: el offset es absoluto.
  - SEEK_CUR: se suma al offset actual.
  - SEEK_END: se suma al offset del final del fichero.

```
off_t lseek(int fd, off_t offset, int whence);
```

Open

- `open`: abre un fichero en el modo indicado por el segundo parámetro. Se usará el primer descriptor de fichero libre.
  Retorna el número del descriptor, -1 en error.
- El segundo parámetro es una combinación de flags:
  - O_CREAT: el fichero se quiere crear si no existe. En este caso, hay que proporcionar el tercer parámetro.
  - O_RDONLY: se va a leer.
  - O_WRONLY: se va a escribir.
  - O_RDWR: se va a leer y escribir.
  - O_TRUNC: si el fichero existe y se abre para escribir (O_RDWR, O_WRONLY), entonces se trunca a longitud 0.
  - O_APPEND: se va a escribir siempre al final del fichero (lseek+write).
  - O_CLOEXEC: se debe cerrar automáticamente si se llama a `exec`.
  - O_DIRECT: se salta la cache.
  - ...

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Open

- Los permisos se comprueban en la apertura.
- El sistema mantiene el **offset** (posición actual) de un fichero que está abierto. El offset se actualiza cada vez que se lee o se escribe en él (empieza a 0).
- Si el proceso abre dos veces el mismo fichero, son dos ficheros abiertos (cada uno con su offset, modo de apertura, etc.).

Open

- Si se crea el fichero, el dueño del fichero será el UID del proceso. Dependiendo del sistema, el grupo puede ser el GID del proceso o el grupo del directorio en el que se crea [1].
- En creación, se abrirá el fichero en el modo indicado, independientemente de los permisos que se le asignen.
- Los permisos efectivos dependen de la máscara...

[1] En linux, depende de si el directorio en el que se crea tiene a 1 el bit `set-group-ID`.

Umask

- `umask` es una propiedad del proceso (como su PID o el directorio de trabajo actual). Se hereda del padre.
- Los bits a 1 en esa máscara no se pondrán en la creación (aunque los indiquemos). En realidad se ponen estos (siendo `mode` el modo que hemos pasado):

```
(mode & ~umask)
```

- P. ej. si `umask` es 077, aunque pongas los permisos al crear a 755, los permisos finales son 700.

Umask

- Podemos poner el `umask` del shell con el comando `umask`.
- `umask`: llamada al sistema que cambia la máscara de creación y devuelve la antigua. Nunca falla.

```
mode_t umask(mode_t mask);
```

Dup

- `dup`: duplica un descriptor de fichero en la primera posición disponible en la tabla. Retorna el número del descriptor en el que se ha duplicado, -1 en error.
- `dup2`: duplica el descriptor en la posición indicada en el segundo parámetro, cerrándolo antes si estaba abierto.

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

Close

- `close`: cierra un fichero abierto y libera los recursos. Retorna -1 si hay un error. Hay que cerrar los ficheros que ya no vayamos a usar. Las tres primeras posiciones en la tabla de descriptores no se deben quedar cerradas: el proceso siempre debe tener entrada, salida y salida de errores.

```
int close(int fd);
```

Access

- `access`: devuelve cero si el fichero puede accederse en el modo indicado, -1 en otro caso.
- Bits para `mode`:
  - F_OK: si existe.
  - R_OK: si se puede leer.
  - W_OK: si se puede escribir.
  - X_OK: si se puede ejecutar.

```
int access(const char *pathname, int mode);
```

Unlink

- `unlink`: elimina un nombre (enlace duro) de un fichero. Si es el último que le queda a un fichero, y ningún proceso lo tiene abierto, el fichero se elimina y se liberan sus recursos.

```
int unlink(const char *pathname);
```

Stat

- `stat`: lee los metadatos de un fichero, los deja en la estructura que se pasar por referencia. Si falla retorna -1.
- Cuidado: atraviesa los enlaces simbólicos. Si queremos los metadatos del enlace y no del fichero apuntado, hay que usar `lstat`.

```
int stat(const char *pathname, struct stat *statbuf);
```

Stat

Campos de `struct stat`:

- st_ino: i-nodo del fichero.
- st_mode: entero con el modo, que son permisos, etc.
- st_nlinks: número de nombres (enlaces duros).
- st_uid: dueño.
- st_gid: grupo.
- st_size: tamaño del fichero.
- st_atime: hora del último acceso al fichero.
- st_mtime: hora de la última modificacón de sus datos.
- st_ctime: hora del último cambio en sus datos o metadatos (excepto en el tiempo de acceso).
- ...

`man 7 inode`

Stat

Aplicando la máscara `S_IFMT` al campo `st_mode` podemos saber el tipo de fichero, comparando con estas constantes:

- S_IFDIR: es un directorio.
- S_IFREG: es un fichero normal.
- S_IFLNK: es un enlace simbólico.
- ...

Hay otras máscaras para acceder a los permisos, etc.
`man 7 inode`

Ejemplo:

```
if(st.st_mode & S_IFMT == S_IFREG)
     printf("it's a regular file!\n");
```

Mkdir

- `mkdir`: crea un directorio. Igual que en el caso anterior, los permisos están sujetos al valor de la máscara `umask`.

```
int mkdir(const char *path, mode_t mode);
```

Opendir

- `opendir`: abre un directorio para lectura. El directorio abierto se representa con el tipo `DIR`. En caso de error devuelve NULL.

```
DIR *opendir(const char *dirname);
```

Readdir

- `readdir`: retorna un puntero a la siguiente entrada de directorio, representada por la estructura `dirent`. Retorna NULL en caso de que no haya más o en error.
- En Linux, la estructura `dirent` tiene estos campos de interés:
  - d_ino: i-nodo.
  - d_name: string con el nombre del fichero.
  - d_type: tipo de entrada de directorio.

```
struct dirent *readdir(DIR *dirp);
```

Closedir

- `closedir`: cierra el directorio. Si no se cierra, tenemos un *leak* de memoria. En caso de error retorna -1.

```
int closedir(DIR *dirp);
```

Otras llamadas al sistema para ficheros

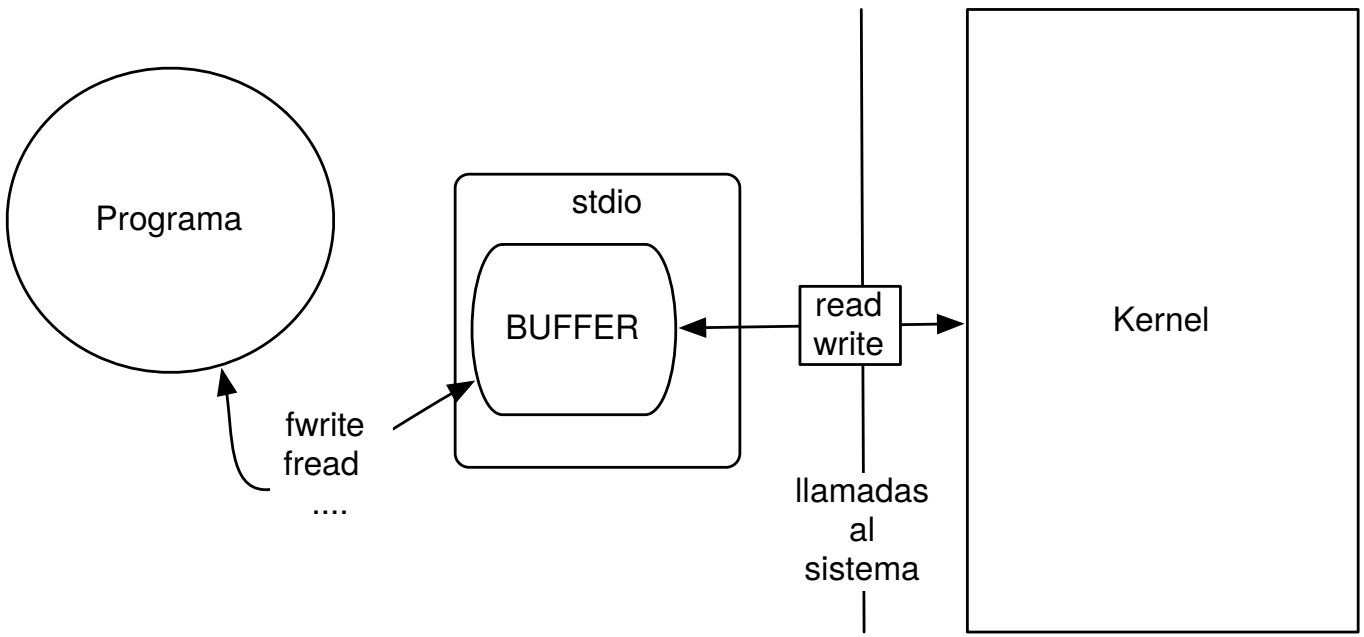
Para modificar los metadatos del fichero:

- `rename`: renombra un fichero.
- `chmod`: cambia los permisos.
- `chown`: cambia el dueño y grupo.
- `utime`: cambia la fecha de acceso y modificación.

Ver páginas de manual (sección 2).

E/S con buffering

- A veces no es fácil leer de un fichero: p. ej. leer líneas de un fichero.
- Realizar una llamada al sistema sale caro: p. ej. leer carácter a carácter.
- Las operaciones de `stdio` ofrecen *buffering* a nivel de la biblioteca (área de usuario).



E/S con buffering

- Stdio usa *streams*, representados por la estructura `FILE`, para representar un fichero abierto.
- Nos ofrece streams para la entrada estándar (`stdin`), salida (`stdout`) y salida de errores (`stderr`).

Fopen

- `fopen`: abre un fichero para hacer E/S con buffering. El modo de apertura es una string, por ejemplo “r” para lecura, “r+” para lectura-escritura, etc.
- `fdopen`: configura un stream a partir de un fichero abierto (de su fd).

```
FILE * fopen(const char *restrict path,
                    const char *restrict mode);
FILE * fdopen(int fildes, const char *mode);
```

Fclose

- `fclose`: vacía el stream, lo cierra y cierra el descriptor de fichero subyacente.

```
int fclose(FILE *stream);
```

Fread

- `fread`: lee del fichero un número de elementos (`nitems`) de tamaño `size`, los guarda a partir de la dirección `ptr`. Retorna el número de elementos leídos. Retorna menos elementos leídos (o cero) que los que se querían leer cuando llega a fin de fichero, pero también cunado tiene un error. Para diferenciar entre esos dos casos, hay que usar las funciones `feof(3)` y/o `ferror(3)`. Tanto esas funciones como `fread` **no actualizan** la variable `errno`.

```
size_t fread(void *restrict ptr, size_t size,
             size_t nitems, FILE *restrict stream);
```

Fwrite

- `fwrite`: escribe en el fichero un número de elementos (`nitems`) de tamaño `size`, de la dirección `ptr`. Retorna el número de elementos escritos.

```
size_t fwrite(void *restrict ptr, size_t size,
              size_t nitems, FILE *restrict stream);
```

Fflush

- `fflush`: fuerza el vaciado del buffer y su escritura en el fichero.

```
int fflush(FILE *stream);
```

Fgets

- `fgets`: lee una línea de hasta `size` bytes. y la guarda en la string `str`. Siempre deja una string. Si no hay más líneas o hay error, retorna NULL.

```
char * fgets(char * restrict str, int size,
                    FILE * restrict stream);
```

Fprintf

- `fprintf`: igual que `printf`, pero escribe en el stream indicado (`printf` siempre lo hace en `stdout`).

```
int fprintf(FILE *stream, const char *format, ...);
```